PolySPIN: Support for Polylingual Persistence, Interoperability and Naming in Object-Oriented Databases

Alan Kaplan
Jack C. Wileden
{kaplan,wileden}@cs.umass.edu

CMPSCI Technical Report 96–4
January 1996

Convergent Computing Systems Laboratory
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This material is based upon work sponsored by Texas Instruments Inc. under grant SRA-2837024. The content does not necessarily reflect the position or policy of Texas Instruments and no official endorsement should be inferred.

Abstract

Heterogeneity in information systems gives rise to a range of interoperability problems. For heterogeneous information systems based on traditional database technology, the dominant interoperability problems are multiple database (or schema) integration and multilingual access to individual databases. While similar interoperability problems can arise in heterogeneous information systems based on object-oriented database (OODB) technology, the nature of this new technology introduces new possibilities for heterogeneity and concomitant new interoperability problems. In particular, a given object-oriented database may contain data objects originally defined, created and persistently stored using the capabilities provided by several distinct programming languages, and an application may need to uniformly process those data objects. We call such a database *polylingual* and term the corresponding interoperability problem the *polylingual access* problem.

While many of today's OODBs support multiple programming language interfaces, none provide transparent polylingual access to persistent data. Instead, present day interoperability mechanisms generally rely on external data definition languages (such as ODMG's ODL), thus reintroducing impedance mismatch and forcing developers to anticipate heterogeneity in their applications, or depend upon direct use of such low-level constructs as the foreign language interface mechanisms provided in individual programming languages.

In this paper we present PolySPIN, an approach supporting polylingual persistence, interoperability and naming for object-oriented databases. We describe our current realization of PolySPIN as extensions to the TI/Arpa Open Object-Oriented Database and give examples demonstrating how our PolySPIN prototype supports transparent polylingual access between C++ and CLOS applications. We also contrast the PolySPIN approach with existing approaches.

1. Introduction

Over the years, as information systems applications have grown larger and more complex, various kinds of *heterogeneity* have appeared in those applications. As a result, individuals and organizations involved in developing, operating or maintaining such applications have increasingly been faced with *interoperability problems* – situations in which components that were implemented using different underlying models or languages must be combined into a single unified application. To aid in overcoming such problems, a range of *interoperability approaches* have been employed. As interoperability problems evolve, due in part to evolution of the underlying models and languages used in information systems applications, interoperability approaches must also evolve.

In applications developed using traditional database technology, there have been two primary sources of heterogeneity. One of these is the need or desire to code different components of an application in different programming languages. The other is the need or desire to make use of two or more different databases in a single application. These have given rise to two corresponding classes of interoperability problems, which we refer to as the *multilingual access* problem and the *multiple database integration* problem.

One of the important extensions to database technology that has appeared during the last decade has been the introduction of object-oriented databases (OODBs). By virtually eliminating impedance mismatch, object-oriented database technology can be viewed as a significant evolution of the underlying models and languages used in information systems applications and hence has many ramifications. Among these are new possibilities for heterogeneity and concomitant new interoperability problems, which necessitate the evolutionary development of new interoperability approaches. In particular, a given object-oriented database may contain data objects originally defined, created and persistently stored using the capabilities provided by several distinct programming languages. We call such a database *polylingual*. This novel kind of heterogeneity induces new interoperability problems, such as the possibility that an application may need to uniformly process the data objects in a polylingual OODB. We term this interoperability problem the *polylingual*

access problem. Existing interoperability approaches provide little or no support for polylingual access, so new approaches must evolve to provide such support.

While many of today's OODBs support multiple programming language interfaces (e.g., ObjectStore [11], GemStone [5]), none provide transparent polylingual access to persistent data. Instead, present day interoperability mechanisms generally rely on external data definition languages (such as ODMG's ODL [7] or CORBA's IDL [16]), thus reintroducing impedance mismatch and forcing developers to anticipate heterogeneity in their applications, or depend upon direct use of such low-level constructs as the foreign language interface mechanisms provided in individual programming languages.

In this paper we focus on the polylingual access problem for object-oriented databases. We begin by discussing heterogeneity and interoperability in OODBs, introducing an example that illustrates various interoperability and heterogeneity issues, identifying some important facets of OODB interoperability problems, particularly the polylingual access problem, and assessing current approaches to OODB interoperability. We then describe PolySPIN, a framework supporting persistence, interoperability and naming for polylingual object-oriented databases, and its current realization as extensions to the TI/Arpa Open Object-Oriented Database [21]. In addition, we show how PolySPIN can facilitate aspects of interoperability in polylingual object-oriented databases, returning to our earlier example to illustrate PolySPIN's capabilities. The paper concludes with a summary, an assessment of our results to date and suggestions of future directions for this research.

2. Heterogeneity and Interoperability in OODBs

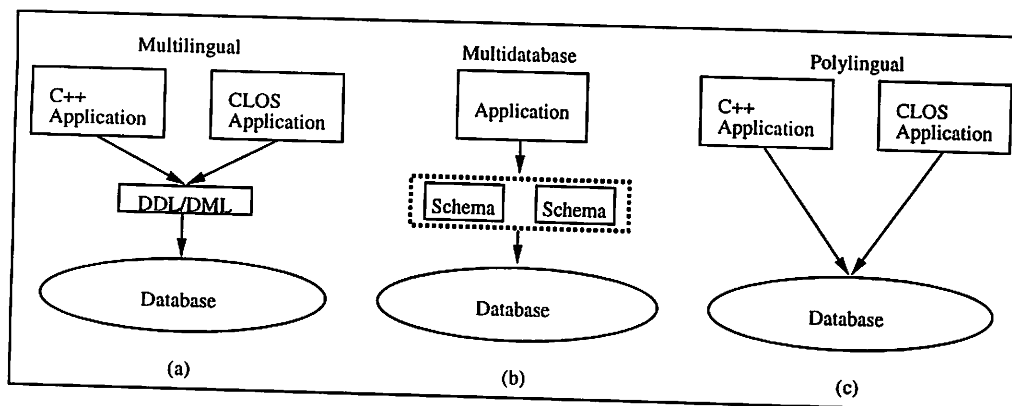

Figure 1: Interoperability Problems in Databases

As noted above, one of the interoperability problems found in applications based on traditional database technology is the multilingual access problem. This problem arises when two or more pieces of application software, written in two or more different languages, need to access the same database, as shown in Figure 1(a). Because traditional database technology has imposed a strong separation between the languages used for defining and manipulating information in a database and the languages in which applications are coded, application developers using traditional databases have been accustomed to dealing with two different type systems – the native type system of the language in which the application is written and the type system embodied in the DDL associated with the database – and two different sets of data manipulation constructs – those of the programming language and the DML, respectively. This dichotomous treatment

of database-resident (persistent) data and program-resident (transient) data has been advantageous with respect to interoperability approaches aimed at addressing the multilingual access problem. In particular, it has meant that simply by defining language-specific bindings between each application programming language and the constructs of the database system's DDL and DML, an interoperability approach that fit smoothly with the programming style already familiar to application developers could be achieved.

The other interoperability problem found in applications based on traditional database technology is the multiple database (or schema) integration problem, as shown in Figure 1(b). This problem arises when two or more different databases are needed by the same application. Approaches to addressing this interoperability problem have primarily involved creating a unified perspective on the multiple databases through some means of reconciling differences in the schemas, or in some cases the internal representations, used in the different databases. Because the multiple database integration problem, in its most general form, requires determination of semantic equivalence among different data type definitions, no single satisfactory solution to this interoperability problem exists. Nevertheless, a number of useful interoperability approaches have been proposed [4], such as various kinds of multidatabases [12] or federated databases [8]. Related work that addresses essentially the same set of concerns from a perspective influenced by the AI view of information (or knowledge) and its processing has proposed such notions as mediators [22] or enabling technology for knowledge sharing [14] to address the schema, model and internal-representation reconciliation facets of the multiple database integration problem.

With the advent of OODBs (which have largely removed the dichotomous treatment of persistent and transient data), the traditional interoperability problems – multilingual access and multiple database integration – remain, but become somewhat more complex. Moreover, object-oriented databases also introduce additional kinds of heterogeneity and concomitant new interoperability problems. In particular, for applications developed using traditional database technology, multilingual access to a single database involves accessing objects that might have been produced by multiple programs written in multiple languages, but which in fact are all defined in terms of a common type model (DDL) and can be manipulated by a common set of operations (DML). With OODBs, however, it is possible to have multiple application programming interfaces (APIs) to the same database, through which objects created using different programming languages (and their respective type models, rather than a common DDL) may all be stored in the same database. The TI/Arpa Open Object-Oriented Database (Open OODB) [21], for instance, provides both a C++ and a Common Lisp Object System (CLOS) API. Moreover, type definitions in an object-oriented model include a set of operations for manipulating instances of the types, so the OODB may well contain operations that are defined and implemented in different programming languages. We use the term ***polylingual database*** to connote this new kind of heterogeneity, in which a single database may contain objects that differ not only in the language used to define their data representations but also in the language used to implement the operations provided by their interface definitions.

Polylingual object-oriented databases, as shown in Figure 1(c), give rise to new interoperability problems. Specifically, in such a database it is possible that even though two objects are of the same type, or equivalent or compatible types,[1] their implementations might be based on different programming languages, posing a serious interoperability problem for an application that wishes to access both of them. Ideally, the differences in their underlying implementation languages should be invisible to the application, which should be able to treat the objects uniformly. In type systems research, a similar goal of making differences in data types transparent to code that processes instances of the types is referred to as polymorphism (e.g., [6]). By analogy, we term this interoperability problem the *polylingual access* problem.

---

[1] Although techniques for determining type equivalence or compatibility are a challenging research topic in their own right (e.g., [23, 24, 25]), for purposes of this paper we presume such determinations can be made, in at least some interesting cases. We discuss this issue further in Section 4..

The remainder of this section is organized as follows. Section 2.1 presents a hypothetical interoperability scenario illustrating the polylingual access problem. Section 2.2 discusses some general interoperability issues arising in OODBs. Section 2.3 concludes with a discussion of existing approaches and how these approaches address the issues presented in the preceding section.

2.1 An Example

As a simple illustration of heterogeneity and interoperability problems in object-oriented databases, consider the following example:

At Hypothetical University, two colleges have independently developed information systems applications, using object-oriented database technology, for managing personnel information regarding their students and faculty. Although both colleges have in fact utilized the same OODB, the Arts College has built their application on a CLOS API while the Sciences College has built theirs on a C++ API. Figure 2 shows a portion of the C++ schema used by the Sciences College,

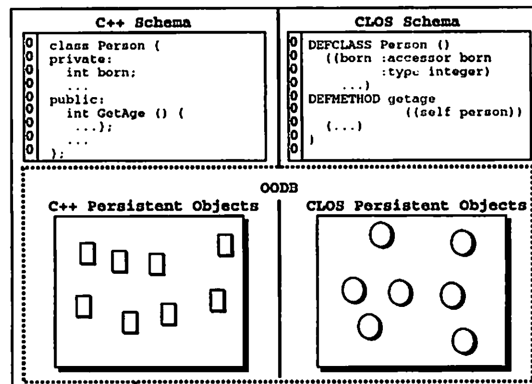

Figure 2: OODB Used By Hypothetical University

a portion of the CLOS schema used by the Arts College, and the OODB containing instances of the personnel data object from both colleges implemented in their respective languages.

The central administration at Hypo U would like to develop some applications making use of personnel information from both colleges. Naturally, they cannot hope to convince either college to translate its personnel information to a representation corresponding to the other's API. Nor can they expect to convince other colleges, when they develop their own personnel information systems in the future, not to use the API of their choosing (e.g., Ada 95 for the Engineering College, Object-Oriented COBOL for the Business College, etc.). Hence the administrators would like their application to be able to be oblivious to the implementation languages of individual persistent objects. They would also like to be able to employ either navigational access or associative access in processing the personnel information from the various colleges. An example of an OQL-style query (based on [20]) that might be part of a C++ application, in this case seeking candidates for early retirement incentives, is shown in Figure 3. Note that the query should be able to be applied to *all* the personnel data resident the OODB, i.e., independent of the language used to create the persistent objects.

```
setPerson = ...; // constains Person objects
Set<Person*> result = NULL;
Query (
  result = SELECT *
        FROM Person* matched IN setPerson
        WHERE matched->GetAge() > 45;
)
```

Figure 3: OQL-style Query

Despite the fact that the two college's personnel information schemas are clearly equivalent, existing OODBs, even those such as the TI/Arpa Open OODB that provide multiple APIs, do not support the kind of polylingual access desired by the Hypo U administrators. Several aspects of current OODB technology stand in the way of polylingual access. We first briefly discuss interoperability goals and issues in general, then specifically consider the approaches to polylingual interoperability that are necessitated by existing OODB interoperability support.

2.2 Interoperability Goals and Issues

Our work on interoperability is, and has been for several years [23], motivated by a primary concern for the impact of an interoperability approach on applications developers. In our view, among the most important objectives for any approach to interoperability are the following:

- Developers should have maximum freedom to define types of objects that their programs manipulate. In particular, they should always be able to use the type systems provided by the language(s) in which they are designing and developing components of their application.
- Whether a data object is to be shared among an application's components should have minimal impact on the components' developers. In particular, making (or changing) a decision about whether, or with what other components, a data object may be shared should not affect the definition of, or interface to, the object.

Given these objectives, we have noted three major sets of issues regarding interoperability in OODB-based applications. Briefly, these are:

Naming How are objects in the persistent store accessed by applications that wish to interoperate through sharing those objects? Current OODBs typically rely on distinct and often incompatible name management mechanisms for each of the programming languages or application programming interfaces (APIs) they support. This results in disjoint persistent stores segregated according to the language used to define the persistent objects and also leads to inconsistent semantics for the name management capabilities provided by the various language interfaces.

Timing When is the decision to share data objects among an application's components made? This question has a dramatic impact on the suitability of different approaches to interoperability. Three distinct timing scenarios for interoperability decisions can be characterized by the relationship between the relative times at which the sharing or shared components are developed and the decision to share them is made, as illustrated in Figure 4. The salient features of each scenario are:

Easiest case: The decision to share is made before any components are developed. In this case, a common (e.g., IDL) description of the shared data objects can be created prior to development

of the components that will share them, language-specific descriptions can be directly created by mapping from the common description, and hence determination of type compatibility is trivial.

Common case: The decision to share is made after one of the sharing components is developed but before any others are. In this case, a common (e.g., IDL) description of the shared data objects can be created by mapping from the language-specific description whose existence predates the sharing decision and then the remaining language-specific descriptions can be directly created by mapping from the common description, so determination of type compatibility is again trivial.

Megaprogramming: The decision to share is made after the sharing components are developed. In this case, common (e.g., IDL) descriptions of the shared data objects can be created by mapping from each of the language-specific descriptions, but determination of type compatibility will then depend upon some kind of comparison of these synthesized descriptions and hence is nontrivial.

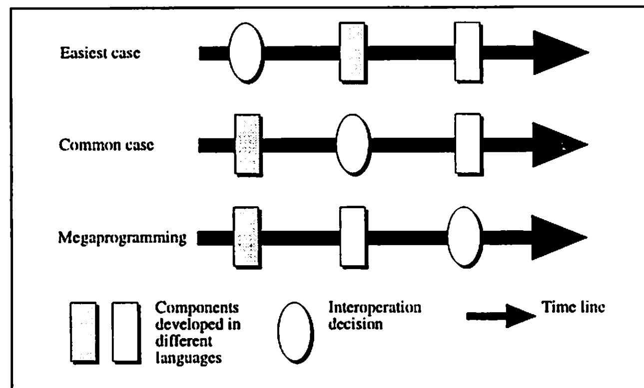

Figure 4: Interoperability Scenarios

Typing How do developers determine whether the types of objects that they wish to share are of compatible types? As noted earlier, this question reduces to the schema integration problem for traditional database technology. For object-oriented database technology, most approaches to addressing this question have been based on use of a unifying type model [23], such as the ODMG ODL. While such approaches may suffice for the easiest and common interoperability scenarios, however, they are inadequate for the megaprogramming case. Since, as our example scenario suggests, that case is perhaps the most important and offers the greatest potential rewards, we have focused our research efforts on attempting to handle it.

2.3 Existing Approaches

Having discussed interoperability issues in general and framed an example scenario with which to illustrate them, we now consider how existing approaches to supporting interoperability in OODB-based applications might address those issues, specifically by examining how an ODMG-based approach might be applied to the Hypo U. situation.

The Object Data Management Group's approach to supporting multilingual applications is based on the use of a (more or less) language-independent unifying type model, expressed in the Object Definition

Language (ODL). ODMG defines bindings between ODL and various programming languages, notably C++ and Smalltalk. The expectation is that, in concert with these bindings, ODL will provide a basis for interoperability in multilingual OODB applications.

Let us look in some detail at how this would apply to the Hypo U. example. As mentioned in our discussion of timing issues related to interoperability, in the easiest and common cases an approach utilizing a unifying type model could work reasonably well. The Hypo U. situation, however, is a paradigmatic example of the megaprogramming case, since the type definitions for the objects that are to be used by the new application have already be formulated in both C++ and CLOS and persistent data instances implemented in both languages have also already been created and stored. How can an ODMG-style approach be made to apply in such an instance?

One possibility would be to proceed to formulate an ODL type definition for the Person type, then attempt to determine whether the ODL definition was consistent with both the C++ and CLOS type definitions for Persons. While automated support for transforming ODL definitions into corresponding type definitions in various programming languages exist, we are unaware of any automated support for going in the opposite direction. Even if such support existed, however, it would leave the (in general undecidable) problem of determining whether the *two* resulting ODL definitions corresponding to the original C++ and CLOS definitions were, indeed, equivalent. Thus, as we noted earlier, the ODMG-style approach offers little assistance with the type compatibility aspect of interoperability problems in the megaprogramming case.

Assuming for the moment, however, that the type compatibility problem could be handled, or ignored, how would the ODMG-style approach be applied in the case of Hypo U.? Unless the developers were so incredibly lucky as to have the C++ and CLOS interface definitions for Person that corresponded to the ODL definition be *identical* to those that had originally been defined and used by the respective college's programmers, the new application would not be able to directly access instances created and stored using those definitions. This would necessitate one of the following:

- Use of "wrapper" code at every point where existing instances were accessed, serving to bridge, or paper over, the discrepancies between the ODL-derived and the original interfaces. This approach essentially negates the usefulness of the unifying model and forces the application to explicitly distinguish between instances created using C++ and CLOS. This clearly falls far short of neatly supporting polylingual application development.

- Transformation of all existing data object instances, either into forms that matched the ODL-derived definitions for their respective languages or into some common (e.g., all C++ or all CLOS) form. While this might solve the problem momentarily, it would not address the possibility that the existing applications might need to be used again in the future, forcing additional transformations on the instances that they might create and store during those future executions. The alternative of modifying the existing applications so that they would create objects according to the ODL-derived definitions might (or might not) be possible, but would likely be error-prone, expensive and in violation of our goal of minimizing the impact of interoperability on the developers of shared, or sharing, components. Hence, use of transformations also fails to make an ODMG-style approach to interoperability acceptable for polylingual programming situations.

Finally, it is worth noting that existing approaches generally require that developers make direct use of mechanisms for multilingual programming, such as foreign-function call-out capabilities, thus forcing explicit recognition of language boundaries and further falling short of supporting polylingual application development.

Most of the shortcomings in existing approaches to interoperability result from inadequate levels of abstraction. By basing higher level abstractions for OODB data object access on a general and uniform

approach to name management, POLYSPIN provides a suitable framework for supporting polylingual access. In the remainder of this paper, we outline this framework and demonstrate its value in overcoming the polylingual access problem.

3. PolySPIN

POLYSPIN is a generic, object-oriented framework that unifies persistence and interoperability capabilities in OODBs from a name management-based perspective, where name management is the means by which a computing system allows names to be established for objects, permits objects to accessed using names, and controls the meaning and availability of names (at any point in time in a particular computation). POLYSPIN, in particular, provides a uniform name management mechanism that not only offers application developers a library of useful abstractions for organizing and navigating object-oriented databases but, as a byproduct, offers an interoperability mechanism providing transparent polylingual access to persistent objects, thus allowing applications to manipulate objects as though they were all implemented in the language of the application. In this section, we begin by briefly describing POLYSPIN's approach to name management. Next, we show how an application developer would use POLYSPIN to enable interoperability in an OODB. The section concludes with a discussion of the internal features of POLYSPIN. Throughout this section, we will refer to the scenario presented in Section 2.1 as a means of explicating various aspects of POLYSPIN. We also note that the example described here has been implemented as extensions to the TI/Arpa Open Object-Oriented Database [21], using Sun C++ [19] and the Lucid Common Lisp Object System (CLOS) [13].

3.1 Name Management and Persistence in PolySPIN

While the benefits of orthogonal persistence capabilities offered by OODBs are widely known, relatively little attention has been to paid to how persistent objects should be organized (from an application's perspective) in an OODB. Typically provided by a name management mechanism, existing approaches in OODBs can be characterized as being relatively *ad hoc* and weak [10]. POLYSPIN addresses these various shortcomings by providing a uniform, flexible and powerful approach to name management. Although the details of its interface are beyond the scope of this paper, the name management mechanism in POLYSPIN allows names to be assigned to objects in binding spaces (where binding spaces are collections of name-object pairs) and names for objects to be resolved in contexts (where contexts are constructed from existing binding spaces). In addition, binding spaces may be assigned names, resulting in the ability to hierarchically organize the name space for objects (similar to directory structures found in almost all modern file systems). Coupled with the persistent store, this approach results in a name-based persistence mechanism where any object (including those in its transitive closure) bound to a name in a binding space reachable from a specially designated root binding space automatically persists.

To participate in this mechanism, an object's class definition must inherit from a common base class, designated the NameableObject class. By inheriting from this class, instances of the subclass can be, among other things, named and resolved using the operations supported by the various abstractions that make up the POLYSPIN name management mechanism. For example, Figure 5 shows a (partial) C++ class definition named Person, a code fragment showing how a name might be assigned to an instance of Person, and a portion of a persistent store organization based on this approach.

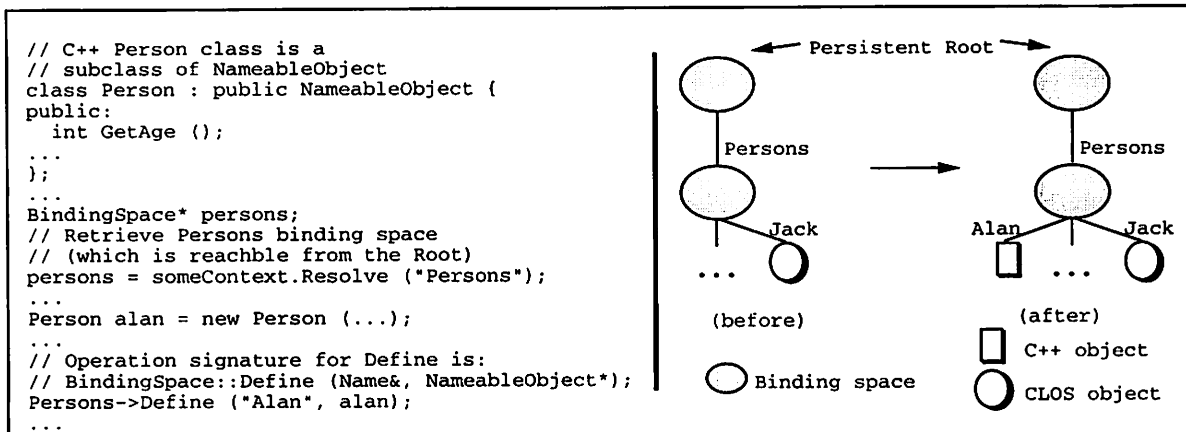

Figure 5: Using PolySPIN's Name Management Mechanism

3.2 Name Management and Interoperability

It is important to realize that making a class inherit from NameableObject can be done, and might frequently be done, quite independently of any decisions to make objects interoperate. That is, however, not its sole purpose. More specifically, inheriting from NameableObject does, in fact, enable the use of the interoperability capabilities of POLYSPIN. First, having a uniform name management mechanism in place results in a language-independent method of establishing visibility paths to persistent objects (i.e., via their assigned names), regardless of the defining language of either the objects or the applications. Second, the name management mechanism serves as a useful place for capturing and recording language-specific information about objects, which can be used to support polylingual access. In particular, once an application has established an initial connection to a persistent object (via its name), the name management mechanism can provide the necessary information permitting the application to create a data path to an object. In other words, when resolving a name of some object (on behalf of some application), the name management mechanism can detect the defining language of the object, and set up the necessary communication medium for manipulating the object. The features supporting this capability are hidden from application developers within the internals of POLYSPIN architecture, which we defer discussion of until Section 3.3.

Given this interoperability mechanism, what is needed is the ability to determine whether two class interfaces defined in different languages can indeed interoperate, and in the event they can, to instrument their implementations (including generating any necessary foreign function interface code) such that the interoperability features of POLYSPIN can be employed. As a step toward automating this process, we have developed a tool called POLYSPINNER. (A more detailed description of POLYSPINNER can be found in [9].) The overall objective of POLYSPINNER is to provide transparent polylingual access to objects with minimal programmer intervention as well as minimal re-engineering of existing source code. The current prototype uses an exact signature matching rule in determining the compatibility between C++ and CLOS classes. It also encapsulates the foreign function interface mechanism for both Sun C++ and Lucid CLOS, as well as the various internal features of POLYSPIN.

To help illustrate how an application developer might use POLYSPINNER, we return to the scenario presented in Section 2.1. In this example, the OODB contains instances of a Person class, where some of the

instances have been developed in C++ and others have been developed in CLOS. To take advantage of the naming facilities offered by PolySPIN, we further assume that the original class definitions for each class already inherit from the NameableObject class, as provided by each language. Thus, prior to any decision to interoperate, the objects resident in the OODB might be organized as shown in the left hand portion of Figure 6. In this scenario, the central administration at Hypo U wished to develop an application supporting queries of the kind shown in the right hand portion of Figure 6. Specifically, the C++ OQL-style query shown here is embedded in a fragment accessing both the C++- and CLOS-defined objects and performing

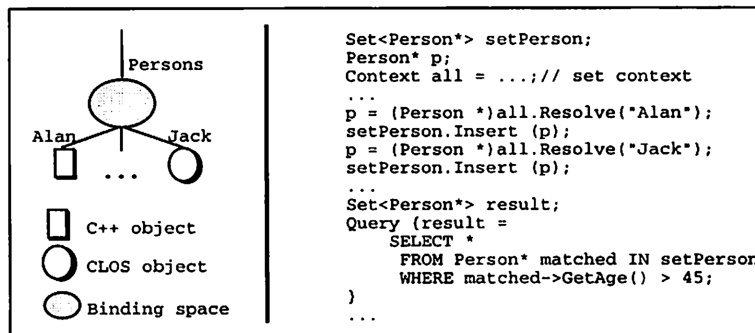

Figure 6: PolySPIN-based OODB

the desired query. Note that the implementation of each object is completely transparent to the C++ OQL-style query, that is, from the application's perspective, both objects are instances of the C++ Person class, even though one is obviously implemented as a CLOS object. To accomplish this, the application developer would take the following steps:

1. Apply PolySPINner to the interfaces and implementations of both the Person C++ and CLOS classes. For example, Figure 7 shows the class definitions for the C++ and CLOS Person classes, where the plain face type represents the original source code and the boldface type represents the code generated by PolySPINner.

   (a) PolySPINner first determines whether or not the two Person classes are compatible by comparing the class interfaces. Using an exact match rule, it should be clear that the C++ and CLOS class interfaces shown in Figure 7 are compatible with one another.

   (b) Since the C++ and CLOS Person classes are deemed compatible, the tool next generates foreign function interface code corresponding to each of the operations associated with each of the classes. This permits calls from C++ to CLOS and vice versa. For example, foreign function interfaces corresponding to each of the "GetAge" operations provided by each of the classes must be generated.

   (c) The tool also modifies the implementations of each of the operations defined by a class. The modifications essentially wrap each operation with switching logic that determines the actual language an object is implemented in and makes the callout to the code generated in the previous step, if need be. For example, if the C++ application invokes the "GetAge" operation on what is in reality a CLOS object, the CLOS "GetAge" operation must be invoked; otherwise the original C++ code implementing the C++ "GetAge" operation must be executed.

2. Re-compile the modified class implementations and the generated source code.
3. Re-link the application.

```
class Person : public NameableObject {
private:
  int born;
public:
  int GetAge ();
};
// GetAge member function
int Person::GetAge () {
  if (this->language == CLOS)
    return
      (__Callout_CLOS_Person_GetAge(this->tidForObject));
  else {
    int result;
    result = 1995 - born;
    return (result);
  }
}
// Callout CLOS Person GetAge
extern "C" int __Callout_CLOS_Person_GetAge (TID this);
// Callout from CLOS into C++
extern "C" int __Callout_CPP_Person_GetAge (TID self ) {
  Person* object = (Person *) TidToCid (self);
  return (object->GetAge());
}
```

C++ Person Class

```
(defclass Person (NameableObject)
  ((born :accessor born
         :type Date
         :initform "MM/DD/YY")
  )
)
;; GetAge method
(defmethod GetAge ((this Person))
 (declare (return-values Integer))
 (cond ( (EQUAL (language this) CLOS)
         (- Today (born this))
       ( (EQUAL (language this) C++)
         (__Callout_CPP_Person_GetAge (tid this)))
 )
);; Callout C++ Person GetAge
(DEF-ALIEN-ROUTINE ("__Callout_CPP_Person_GetAge"
                    __POLYSPIN_CPP_Person_GetAge)
 int (self TID )
)
;; Callout from C++ into CLOS
(DEF-FOREIGN-CALLABLE
            (__Callout_CLOS_Person_GetAge
            (:language :c) (:return-type int))
            ( ( this TID) )
 (GetAge (tid-to-cid this))
)
```

CLOS Person Class

Figure 7: Results of Applying PolySPINner

As should be evident, neither the class interfaces nor the persistent data are modified by the PolySPINner tool. Only the class implementations must be re-compiled, along with the generated source code. In addition, the original application remains unchanged, although it must be re-linked to accommodate the changes made to the class implementations. Note that, in Figure 7, some of the PolySPINner generated code contains references to *CIDs* and *TIDs*. Although transparent to applications, these resources enable polylingual access in PolySPIN. In the remainder of this section, we describe these and other internals of PolySPIN that enable polylingual access.

3.3 The Internal Features of PolySPIN

The fact that objects themselves may be implemented in different languages is completely hidden within PolySPIN's name management mechanism. To support this level of transparency in applications, the PolySPIN framework utilizes the following components:

- A three-level object identifier hierarchy.
- A common base class encapsulating language-specific information for transient objects.
- A universal object representation encapsulating language-specific information for persistent objects.

As we illustrate in the remainder of this section, these abstractions, together with their interactions with one another, form a suitable foundation for providing transparent polylingual access.

3.3.1 The Object Identifier Hierarchy

A common solution to the interoperability involves converting between data representation formats. For example, to achieve interoperability in the scenario described earlier, it might be possible to simply translate C++ Person objects into CLOS objects (and vice versa). Unfortunately, even when hidden from users and applications, such techniques can be prone to error and computationally expensive, especially for large and complex objects.

An alternative approach involves utilizing object references (or L-values) for identifying objects. This solution has the obvious benefits in terms of efficiency and maintainability; one drawback, however, is that different programming languages use distinct and incompatible object reference mechanisms. For example, native references to objects in C++ can not be interchanged with references to CLOS objects (and vice versa). Instead, a distinct mechanism must be used in a CLOS application to identify a C++ object. Languages supporting garbage collection, e.g., CLOS, present further complications since the value of an object identifier may change over the course of a computation. Although transparent to CLOS applications, garbage collection may cause subsequent accesses by a C++ application using a native CLOS object identifier to result in invalid or dangling references. The addition of persistence yields yet another identifier mechanism that must be managed despite the fact that persistent identifiers are generally hidden from applications. More specifically, when an object is designated as being persistent, a persistent identifier is assigned to the object, where the persistent identifier is typically bound to some user-level name. When the object is retrieved from the database, the persistent identifier is first used to locate the object. A reference (i.e., an L-value) must then be created for the object so that the application can access and manipulate the object.

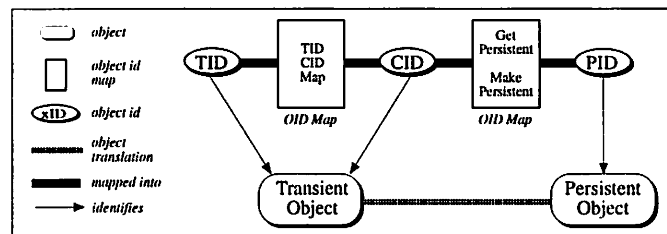

Figure 8: PolySPIN's Three Levels of Identifiers

As a step toward relieving application developers from managing separate object identifier mechanisms or building special-purpose ones, the PolySPIN framework maintains a three-level identifier hierarchy, as shown in Figure 8. The hierarchy, in order of increasing lifetime, consists of:

- A *computation identifier* (or CID), which is an L-value or reference used by applications for identifying, accessing and manipulating objects defined in the same language. A CID for a particular object may change over the the course of a computation, although such changes, for all intents and purposes, are invisible to programmers.
- A *transient identifier* (or TID), which is an active-computation-unique identifier for an object. Once assigned during some active computation, it is assumed that a TID does not change over the course of that computation's lifetime.

- A *persistent identifier* (or PID), which is a globally unique identifier for a persistent object. When a object is made persistent, a PID is assigned to the object. A PID is assumed to be immutable over the course of the object's lifetime.

In addition, for each language, PolySPIN provides two-way TID↔CID and CID↔PID mapping mechanisms, where the former map can be implemented using a traditional hash table, and the latter map is often supplied by the underlying persistence mechanism provided by the OODB. PolySPIN's maintenance of these identifiers, along with functions for mapping between them, means that applications simply use CIDs to manipulate objects. Any required identifier translations are handled by PolySPIN. For example, when an application retrieves an object from the persistent store (i.e., via name resolution), a CID identifying the object is returned to the application. As we will show in the following sections, the CID points to an object that, from the application's point of view, looks and behaves as if the object were defined in the same language as the application.

3.3.2 The NameableObject Class

As noted earlier, inheriting from the NameableObject class offers applications developers the ability to use PolySPIN's improved name management mechanism. At the same time, the NameableObject class encapsulates various language-specific information for an object including a defining language, a TID, and various type-related information. As shown in Figure 9, values for this information can be computed when

```
// NameableObject class
class NameableObject {
private:
  LanguageId language;
  TID tidForObject;
  ClassId classInfo;
public:
// Constructor for NameableObject
  NameableObject() {
    language = C++;
    tidForObject = CidToTid (this);
    ...
}
// Constructor for Person class
Person::Person () {
// Implicdy invokes NameableObject
// Then set type information
  classInfo = "Person";
  ...
}
```

Figure 9: The NameableObject Class

an object (derived from NameableObject) is instantiated. For example, the constructors for the C++ Person and NameableObject classes in Figure 9 illustrate how this information is computed and recorded. (The same information is computed in an analogous fashion for CLOS.)

When an operation is invoked on an object, the data maintained by the NameableObject can be used to determine the actual implementation of an object. Since this is hidden from users, however, all instances of the class can be viewed and accessed through a single language interface, even though various instances may in fact be implemented in various languages. Returning to the C++ and CLOS classes shown in Figure 9, the "GetAge" operation for the C++ Person class first checks the value of the defining language for the object. If the object is implemented in C++, then the C++ implementation of the "GetAge" operation is used. If, on the other hand, the object is implemented in CLOS, then the corresponding CLOS operation must be

invoked (on the CLOS object). This involves making a call-out to a CLOS function (in this example, using the foreign function interface mechanisms of C++ and CLOS) and passing the CLOS object's TID and the value for increment parameter. On the CLOS side, the TID is first mapped into its CID value and then the actual CLOS "GetAge" operation is invoked.

3.3.3 The Universal Object Representation

PolySPIN unifies the persistent store by permitting the co-existence of objects implemented in different languages. Furthermore, access to the persistent store is provided by a name management mechanism that is uniformly available across multiple programming languages. To support polylingual access to persistent objects, PolySPIN introduces a level of indirection in bindings between names and (persistent) objects called a universal object representation or UOR.

Like the NameableObject class, a UOR encapsulates various language-specific information about objects, including an object's PID. A UOR is created for an object when that object is assigned a name. In particular, the information stored by the NameableObject is transferred to the UOR. If the object is later designated as being persistent (as described above), then a value for the object's PID is also set in the UOR. Later, when the object is accessed (by resolving the name of the object), the name management mechanism can use the information stored in the UOR to return an appropriate object to the application.

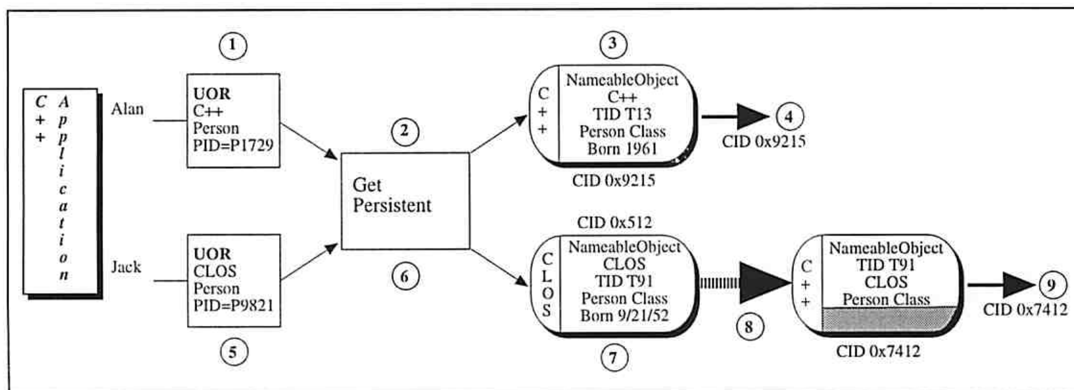

Figure 10: Accessing Objects Via UORs

For example, Figure 10 shows the various tasks PolySPIN performs on behalf of a C++ application accessing two instances of the Person class, where one object, bound to the name "Alan," is implemented in C++ and the other, bound to the name "Jack," is implemented in CLOS. When the object named "Alan" is accessed, PolySPIN examines the UOR for the object (1), determining the object's defining language and its PID. Based on the information, the object is retrieved from the store (2) and a transient, C++ version is constructed (3). Since the language of the application and the object are the same, the object (i.e., its CID) is simply returned to the application (4). When the object named "Jack" is accessed, PolySPIN again examines the UOR for the object (5), determining the object's defining language and its PID. Based on the information, the object is retrieved from the store (6) and a transient, CLOS version is constructed (7). Since the language of the application and object are in this case different, PolySPIN creates a C++ Person object (8), which acts as a "surrogate" for its corresponding CLOS version. The surrogate's defining

language is set to "CLOS" and its TID is set to the TID of the CLOS object. The rest of the data associated with the C++ Person surrogate is simply ignored (as indicated by the shaded portion of the surrogate object in the figure). Finally, the CID of the surrogate is returned to the application (9).

Subsequent accesses to both objects will (eventually) invoke the implementing object (as described above in Section3.3.2). For example, the query shown in Figure 6 calls the "GetAge" operation for both objects. As shown in Figure 7, for the object named "Alan," the original C++ "GetAge" operation is called, while for the object named "Jack," the CLOS "GetAge" operation is invoked. Thus, the query is able to access and process both objects, despite the fact that one object is implemented in C++ and the other in CLOS.

4. Conclusion

In this paper, we have described a new class of interoperability problem for OODBs, namely the polylingual access problem. We have also described POLYSPIN, an approach supporting persistence, interoperability and naming in OODBs, and we have shown how POLYSPIN can be used to overcome the polylingual access problem in OODBs. We have briefly described POLYSPINNER, a tool to help automate the use of POLYSPIN and illustrated its capabilities using a simple, but representative, example of a polylingual OODB application. Finally, we have discussed how our approach has been realized in a prototype implementation of POLYSPIN and POLYSPINNER supporting polylingual access between C++ and CLOS, built as an extension to the TI/Arpa Open OODB.

At various points in this paper we have alluded to other approaches to supporting heterogeneity in OODBs that have been proposed, primarily those advanced by the Object Management Group (OMG) and the Object Data Management Group (ODMG). Each group has proposed an object-oriented type model (IDL [16] and ODL [7], respectively) that could be used, in effect, to re-introduce some of the dichotomy between the type models of application programming languages and those of object-oriented databases (as discussed in Section 2.). For the multilingual access problem, the widespread adoption of IDL or ODL (which is a superset of IDL) would have the effect of reducing the number of bindings needed, since the availability of a binding from any given programming language to IDL (ODL) would suffice to support multilingual access. Because both IDL and ODL are quite rich object-oriented type models, however, this approach would not overcome the difficulty of creating the bindings, reduce their complexity or contribute to making them more exact. Similarly, the multiple database integration problem could be somewhat ameliorated by IDL or ODL, which would provide a common basis for attempts to reconcile difference between OODB schemas, but would not significantly reduce the inherently greater complexity of determining semantic equivalence between object-oriented type definitions.

Experimentation with POLYSPIN has suggested a number of future research directions. In our view, the most significant shortcoming in our initial prototypes is the very limited notion of type compatibility that they embody. Specifically, the determination of type compatibility embedded in POLYSPINNER is essentially an instance of *exact signature matching*, the most restrictive category in the Zaremski and Wing classification of signature matching approaches [24]. While we intentionally adopted this restrictive definition of type compatibility for our initial prototypes so as to focus on other aspects of supporting interoperability for polylingual systems, and even though a number of interesting examples can be successfully handled despite this limitation, we are eager to expand the type compatibility aspect of our approach. As a first step, we are currently investigating how the more general categories of signature matching techniques in the Zaremski and Wing classification can be brought to bear on multilingual and polylingual interoperability situations, and we are beginning to develop automated support for doing so. This will dramatically strengthen the compatibility checking component of the POLYSPINNER, allowing it to produce much richer and more powerful class compatibility mappings, and thus greatly expanding the applicability of our approach. Specifically, it will

mean that components can interoperate not only by sharing data objects whose types are identical, but also by sharing data objects whose types are related by much weaker, and therefore probably more prevalent and hence more useful, kinds of compatibilities. Subsequently, we plan to investigate richer forms of type compatibility, based on deeper, semantics-based analyses of the relationships between types. Zaremski and Wing's work on *specification matching* [25] and Blaine and Goldberg's work on axiomatic approaches [3] represent two valuable points of departure for our research in this area. In addition, we are currently exploring augmenting Open OODB with other languages, such as Ada95. The addition of other languages will allow us to continue our experimentation with PolySPIN, in an effort to further confirm, or find ways to extend, its generality and applicability.

An important problem closely related to interoperability is the problem of type evolution in persistent object systems [2, 1, 15, 17, 18]. Indeed, the type evolution problem can be seen as a special case of interoperability, one in which the need for interoperation arises due to changes in the definitions of one or more data types that have existing persistent instances. We plan to investigate the extension of our interoperability approach to the type evolution problem, in particular how enhanced versions of the PolySPINner tool can be brought to bear on this problem.

Finally, in our initial prototypes we have completely ignored questions of performance and optimization in polylingual OODBs. Clearly, crossing language boundaries can be very expensive. For example, suppose a query is applied to a set containing 10000 objects, half of which are actually implemented in C++ and half of which are actually implemented in CLOS. A naive approach might process the query on an object-by-object basis, possibly resulting in a large number of language boundary crossings. A relatively simple optimization technique might cluster the objects according to their implementing language, perform the query on each subset in each language's address space, and then combine the results. Having demonstrated the potential value of our approach to polylingual access, we now plan to investigate these topics with the aim of improving the approach's practicality.

We believe the work reported in this paper represents an important extension to object-oriented database technology. While modern OODBs often provide multiple language interfaces, interoperating among the various languages can be a cumbersome and complex process, thus limiting their overall potential. PolySPIN provides transparent, polylingual access to objects (of compatible types), even though the objects may have been created using different programming languages. Thus, application developers are free to work in their native language without precluding the possibility of interoperating with foreign language objects or applications.

REFERENCES

[1] J. Banerjee, W. Kim, H.-J. Kim, and H. K. Forth. Semantics and implementation of schema evolution in object-oriented languages. In *Proceedings of the ACM SIGMOD Annual Conference*, pages 311–322, San Francisco, CA, July 1987.

[2] G. Barbedette. Schema modifications in the LISPO_2 persistent object-oriented language. In *Proceedings of the Fifth European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, Geneva, Switzerland, July 1991.

[3] L. Blaine and A. Goldberg. Interoperability of abstract data values. Technical Note 6, DARPA Module Interconnection Formalism Working Group, Jan. 1991.

[4] M. Bright, A. Hurson, and S. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, 25(3):50–60, Mar. 1992.

[5] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, Oct. 1991.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[7] R. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1993.

[8] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[9] A. Kaplan and J. C. Wileden. PolySPINner 1.0: Automating support for interoperability in polylingual software systems. In Preparation.

[10] A. Kaplan and J. C. Wileden. Name management and object technology for advanced software. In *International Symposium on Object Technologies for Advanced Software*, number 742 in Lecture Notes in Computer Science, pages 371–392, Kanazawa, Japan, Nov. 1993.

[11] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, Oct. 1991.

[12] W. Litwin and A. Abdellatif. Multidatabase interoperability. *IEEE Computer*, 19(12):10–18, Dec. 1986.

[13] Lucid, Inc., Menlo Park, CA. *Lucid Common Lisp/Sun: Advanced User's Guide*, version 4.0 edition.

[14] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[15] E. Odberg. MultiPerspectives: The classification dimension of schema modification management for object-oriented databases. In *Proceedings of TOOLS USA*, Santa Barbara, CA, Aug. 1994.

[16] OMG. Object management architecture guide, revision 2.0. OMG TC Document 92.11.1, Object Management Group, Framingham, MA, Sept. 1992.

[17] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 111–117, Orlando, FL, Oct. 1987.

[18] A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 483–495, Portland, OR, Sept. 1986.

[19] Sun Microsystems, Inc., Mountain View, CA. *SPARCompiler C++ 3.0.1 Programmer's Guide*, revision a edition, Oct. 1992.

[20] Texas Instruments, Inc., Dallas, TX. *Open OODB Query Language User Manual*, release 0.2 (alpha) edition, 1993.

[21] D. L. Wells, J. A. Blakely, and C. W. Thompson. Architecture of an open object-oriented management system. *IEEE Computer*, 25(10):74–82, Oct. 1992.

[22] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, Mar. 1992.

[23] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr. Specification level interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.

[24] A. M. Zaremski and J. M. Wing. Signature matching, a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2), Apr. 1995.

[25] A. M. Zaremski and J. M. Wing. Specification matching of software components. In *The Third Symposium on the Foundations of Software Engineering*, pages 6–17, Washington, D.C., Oct 1995.